Implementing Atomic Actions in Ada 95

Andy Wellings and Alan Burns, *Member*, *IEEE Computer Society*

Abstract—Atomic actions are an important dynamic structuring technique that aid the construction of fault-tolerant concurrent systems. Although they were developed some years ago, none of the well-known commercially-available programming languages directly support their use. This paper summarizes software fault tolerance techniques for concurrent systems, evaluates the Ada 95 programming language from the perspective of its support for software fault tolerance, and shows how Ada 95 can be used to implement software fault tolerance techniques. In particular, it shows how packages, protected objects, requeue, exceptions, asynchronous transfer of control, tagged types, and controlled types can be used as building blocks from which to construct atomic actions with forward and backward error recovery, which are resilient to deserter tasks and task abortion.

Index Terms—Software fault tolerance, atomic actions, Ada 95, exception handling, recovery blocks, conversations.

1 Introduction

Techniques for tolerating software faults are often classified according to whether they are static (masking) or dynamic. With static redundancy, several versions of a software component are written and each version executes in response to all requests; voting is performed on the output to determine which result to use. It is static because each version of the software has a fixed relationship with every other version and the voter; and because it operates whether or not faults have occurred. With dynamic redundancy, the redundant components only come into operation *when* an error has been detected.

Dynamic fault tolerance has four constituent phases [1].

1) *Error detection*. Faults of significance will eventually manifest themselves in the form of errors; no fault tolerance schemes can be utilized until errors are detected.
2) *Damage confinement and assessment*. When an error has been detected, a decision must be made on the extent to which the system has been corrupted; the delay between a fault occurring and the manifestation of the associated error means that erroneous information could have spread throughout the system.
3) *Error recovery*. Error recovery techniques aim to transform a corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality).
4) *Fault treatment and continued service*. An error is a symptom of a fault; although the damage may have been repaired, the fault may still exist and, therefore, the error may recur unless some form of maintenance is undertaken.

This paper is primarily concerned with dynamic redundancy techniques and, in particular, damage confinement and error recovery. For sequential systems, damage confinement is well understood; techniques such as modular programming and object-oriented encapsulation (within the context of a strongly typed programming language) enable faults to be confined. Judicious placement of acceptance tests or assertions allow errors to be detected before damage can be propagated. Similarly, techniques such as exception handling [2] (forward error recovery) and recovery blocks [3] (backward error recover) allow error recovery to be performed according to whether the fault was anticipated or not.

For concurrent systems, the position is not so clear cut. Although techniques such as conversations [4] and atomic actions [5] were developed some time ago, few of the mainstream languages or operating systems provide direct support [6]. Instead, languages such as Concurrent Pascal have been used as the basis for experimentation [7], or a set of procedural extensions or object extensions have been produced. For example, Arjuna uses the latter approach to provide a transaction-based toolkit for C++ [8].

Arguably, high-level support is not needed and the required functionality can be programmed with lower-level primitives. For example, some attempts have been made to program conversations in Ada 83 [9], [10], [11]; however, these were severely hampered by the lack of suitable language support. For instance, Romanovsky and Strigini [11] only allow parallelism to exist inside a conversation; the approach is not appropriate if a collection of pre-existing tasks wish to participate collectively in a conversation. None of these attempts address how to structure atomic actions with both forward and backward error recovery in Ada 95.

The goals of this paper are to:

- summarize software fault tolerance techniques for concurrent systems
- evaluate the Ada 95 programming language [12] from the perspective of its support for software fault tolerance
- show how Ada 95 can be used to implement software fault tolerance techniques.

Section 2 reviews the requirements for atomic action and Section 3 briefly describes how forward and backward error recovery can be undertaken. Section 4 describes the new

• *A. Wellings and A. Burns are with the Real-Time Systems Research Group, Department of Computer Science, University of York, Heslington, York YO1 5DD UK. E-mail: {andy, burns}@minster.york.ac.uk.*

Manuscript received Dec. 6, 1996; revised Feb. 12, 1997.
Recommended for acceptance by J.C. Knight.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95104.

features of Ada 95 that are required to implement atomic actions. Section 5 then shows how these features can be used to program: simple actions, actions with backward error recovery, actions with forward error recovery, actions which are resilient to deserters, nested actions, and reusable actions. Section 6 gives a simple example, while Conclusions are presented in Section 7.

2 Atomic Actions

One of the main motivations for introducing concurrent processes into a system is that they enable parallelism in the real world to be reflected in application programs. This enables such programs to be expressed in a more natural way and leads to the production of more reliable and maintainable systems. However, concurrent processes create many new problems which do not exist in the purely sequential program. In particular, consideration has to be given to the way in which groups of cooperating concurrent processes should be structured in order to coordinate their activities. For example, withdrawal from a bank account may involve a ledger process and a payment process in a sequence of communications to authenticate the drawer, check the balance and pay the money. Furthermore, it may be necessary for more than two processes to interact in this way to perform the required action. In all such situations, it is imperative that the processes involved see a consistent system state. With concurrent processes, it is all too easy for groups of processes to interfere with one another.

An atomic action has been proposed as a dynamic mechanism for controlling the joint execution of a group of processes such that their combined operation appears as an *indivisible* action.

There are several almost equivalent ways of expressing the properties of an atomic action [5], [13].

1) An action is atomic if the processes performing it are not aware of the existence of any other active process, and no other active process is aware of the activity of the processes during the time the processes are performing the action.
2) An action is atomic if the processes performing it do not communicate with other processes while the action is being performed.
3) An action is atomic if the processes performing it can detect no state change except those performed by themselves, and if they do not reveal their state changes until the action is complete.
4) Actions are atomic if they can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

Although an atomic action is viewed as being indivisible, it can have an internal structure. To allow modular decomposition of atomic actions, the notion of a *nested atomic action* is introduced. The processes involved in a nested action must be a subset of those involved in the outer level of the action. If this were not the case, a nested action could smuggle information concerning the outer-level action to an external process. The outer-level action would then no longer be indivisible.

2.1 Requirements for Atomic Actions

The implementation requirements for atomic actions are independent of the notion of a process and the form of interprocess communication provided by a language [14]. They are:

1) *Well-defined boundaries*. Each atomic action should have a start, end and a side boundary. The start boundary is the location in each process involved in the atomic action where the action is deemed to start. The end boundary is the location in each process involved in the atomic action where the action is deemed to end. The side boundary separates those processes involved in the atomic action from those in the rest of the system.
2) *Indivisibility*. An atomic action must not allow the exchange of any information between the processes active inside the action and those outside (resource managers excluded). If two atomic actions do share data then the value of that data after the atomic actions is determined by the strict sequencing of the two actions in some order.

   There is no implied synchronization at the start of an atomic action. Processes can enter at different times. However, there is an implied synchronization at the end of an atomic action; processes are not allowed to leave the atomic action until all processes are willing and able to leave.
3) *Nesting*. Atomic actions may be nested as long as they do not overlap with other atomic actions. Consequently only strict nesting is allowed.
4) *Concurrency*. It should be possible to execute different atomic actions concurrently. One way to enforce indivisibility is to run atomic actions sequentially. However, this could seriously impair the performance of the overall system and therefore should be avoided. Nevertheless, the overall effect of running a collection of atomic actions concurrently must be the same as that which would be obtained from serializing their executions.
5) *Recovery*. As it is the intention that atomic actions should form the basis of damage confinement, they must allow recovery procedures to be programmed.

Executing an atomic action requires the participating processes to coordinate their activities. The imposed synchronization on the action is as follows. Processes entering the action are not blocked. A process is only blocked inside the action if it has to wait for a resource to be allocated, or if it attempts to communicate with another process inside the action and that process is either: active in the action but not in a position to accept the communication, or is not, as yet, active in the action.

Processes may leave the action only when all processes active in the action wish to leave. Hence, it is possible for a subset of the named processes to enter the action and subsequently leave (without recourse to any interactions with the missing processes). This facility is deemed to be essential in a real-time system where deadlines are important. It solves the *deserter* problem where all processes are held in an action because one process has not arrived. This will be considered along with error recovery in the next two sections.

3 Recoverable Atomic Actions

This section considers atomic actions with either backward or forward error recovery. Backward error recovery relies on restoring the system to a safe state previous to that in which the error occurred. In contrast, forward error recovery attempts to continue from an erroneous state by making selective corrections to the system state.

3.1 Atomic Actions and Backward Error Recovery

When backward error recovery is applied to groups of communicating processes, it is possible for all the processes to be rolled back to the start of their execution. This is the so called *domino effect*. The problem occurs if there is no consistent set of recovery points or a recovery line. An atomic action provides that recovery line automatically. If an error occurs inside an atomic action then the processes involved can be rolled back to the start of the action and alternative algorithms executed; the atomic action ensures that processes have not passed any erroneous values through communication with processes outside the action. When atomic actions are used in this way they are called *conversations* [15].

With conversations, each action statement contains a recovery block. For example, the following illustrates a component of a three process conversation. This component will be executed by process P_1; P_2 and P_3 will execute similar structures:

```
action A with (P2, P3) do
   ensure <acceptance test>
   by
      -- primary module
   else by
      -- alternative module
   else by
   -- alternative module
   else error
end A;
```

The basic semantics of a conversation can be summarized as follows:

- On entry to the conversation, the state of a process is saved. The set of entry points forms the recovery line.
- While inside the conversation, a process is only allowed to communicate with other processes active in the conversation and general resource managers. As conversations are built from atomic actions, this property is inherited.
- In order to leave the conversation, all processes active in the conversation must have passed their acceptance test. If this is the case then the conversation is finished and all recovery points are discarded.
- *If any process fails its acceptance test, all processes have their state restored to that saved at the start of the conversation and they execute their alternative modules*. It is, therefore, assumed that any error recovery to be performed inside a conversation *must* be performed by *all* processes taking part in the conversation.
- Conversations can be nested, but only strict nesting is allowed.
- If all alternatives in the conversation fail then recovery must be performed at a higher level.

It should be noted that in conversations, as defined by Randell [15], all processes taking part in the conversation must have entered the conversation before any of the other processes can leave. This differs from the semantics described here. If a process does not enter into a conversation, either because of tardiness or because it has failed, then as long as the other processes active in the conversation do not wish to communicate with it then the conversation can complete successfully. If a process does attempt to communicate with a missing process then it can either block and wait for the process to arrive or it can wait for a defined time interval and then continue. Adopting this approach has two benefits [16]:

1) It allows conversations to be specified where participation is not compulsory.
2) It allows processes with deadlines to leave the conversation, continue, and if necessary take some alternative action.

Conversations have been discussed by Kim [7] in the context of extensions to Concurrent Pascal and Tyrrell and Holding [17], and Jalote and Campbell [18], [19] in the context of CSP.

Although conversations allow groups of processes to coordinate their recovery, they have been criticized. One important point is that when a conversation fails, all the processes are restored and all enter their alternative modules. This forces the same processes to communicate again to achieve the desired effect; a process cannot break out of the conversation. This may not be what is required. Gregory and Knight [16] point out that in practice when one process fails to achieve its goal in a primary module through communication with one group of processes, it may wish to communicate with a completely new group of processes in its secondary module. Furthermore, the acceptance test for this secondary module may be quite different. There is no way to express these requirements using conversations. To overcome some of the problems associated with conversations, Gregory and Knight [16] have proposed an alternative approach to backward error recovery with concurrent processes.

3.2 Atomic Actions and Forward Error Recovery

Although backward error recovery enables recovery from unanticipated errors, it is difficult to undo any operation that may have been performed in the environment in which the system operates. Consequently, forward error recovery and exception handling must also be considered. In this section, exception handling between the concurrent processes involved in an atomic action is discussed.

With backward error recovery, when an error occurs all processes involved in the atomic action participate in recovery. The same is true with exception handling and forward error recovery. If an exception occurs in one of the processes active in an atomic action then that exception is raised in *all* processes active in the action. The exception is said to be *asynchronous* as it originates from another process. The following is a possible Ada-like syntax for an atomic action supporting exception handling. As with conversations this is executed by process P_1 with processes P_2 and P_3 executing similar structures.

```
action A with (P2, P3) do
   --the action
exception
   when exception_Y =>
       -- sequence of statements
   when exception_Z =>
       -- sequence of statements
   when others =>
      raise atomic_action_failure;
end A;
```

With the termination model of exception handling, if all processes active in the action have a handler and all handle the exception without raising any further exception, then the atomic action completes normally. If a resumption model is used, once the exception has been handled, the processes active in the atomic action resume their execution at the point where the exception was raised.

With either model, if there is no exception handler *in any one of the processes active in the action* or one of the handlers fails then *the atomic action fails* with a standard exception *atomic_action_failure*. This exception is raised in all the involved processes.

There are two issues which must be considered when exception handling is added to atomic actions: resolution of concurrently raised exceptions and exceptions in nested actions [4]. These are now briefly reviewed.

3.2.1 Resolution of Concurrently Raised Exceptions

It is possible for more than one process active in an atomic action to raise different exceptions at the same time. As Campbell and Randell [4] point out, this event is likely if the errors resulting from some fault cannot be uniquely identified by the error detection facility provided by each component of the atomic action. If two exceptions are simultaneously raised in an atomic action then there may be two separate exception handlers in each process. It may be difficult to decide which one should be chosen. Furthermore, the two exceptions in conjunction constitute a third exception which is the exception which indicates that both the other two exceptional conditions have occurred.

In order to resolve concurrently raised exceptions, Campbell and Randell propose the use of an *exception tree*. If several exceptions are raised concurrently then the exception used to identify the handler is that at the root of the smallest subtree that contains all the exceptions (although it is not clear how to combine any parameters associated with this exception). Each atomic action component can declare its own exception tree; the different processes involved in an atomic action may well have different exception trees.

3.2.2 Exceptions and Internal Atomic Actions

Where atomic actions are nested, it is possible for one process active in an action to raise an exception when other processes in the same action are involved in a nested action. Fig. 1 illustrates the problem.

When the exception is raised, all processes involved must participate in the recovery action. Unfortunately, the internal action, by definition, is indivisible. To raise the exception in that action would potentially compromise that indivisibility. Furthermore, the internal action may have no knowledge of the possible exception that can be raised.

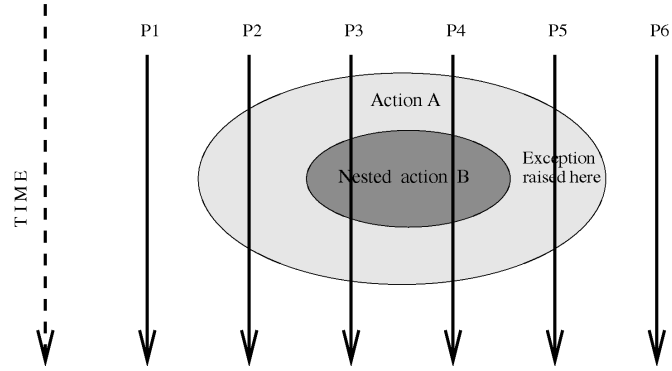


Fig. 1. An exception in a nested atomic actions.

Campbell and Randell [4] have discussed two possible solutions to this problem. The first solution is to hold back the raising of the exception until the internal action has finished. This they reject because:

- In a real-time system the exception being raised may be associated with the missing of a deadline. To hold back the recovery procedure may seriously place in jeopardy the action's timely response.
- The error condition detected may indicate that the internal action may never terminate because some deadlock condition has arisen.

For these reasons, Campbell and Randell allow internal actions to have a predefined abortion exception. This exception is raised to indicate to the action that an exception has been raised in a surrounding action and that the preconditions under which the action was invoked are no longer valid. If such an exception is raised, the internal action should invoke fault-tolerance measures to abort itself. Once the action has been aborted, the containing action can handle the original exception.

If the internal action cannot abort itself then it must signal an atomic action failure exception. This may then be combined with the outstanding exception so as to affect the choice of recovery to be performed by the surrounding action. If no abortion exception is defined, the surrounding action must wait for the internal action to complete. Alternatively, a default handler could be provided which would raise the atomic action failure exception.

4 Ada 95

Ada 83, along with most concurrent programming languages, was unable to support the full functionality of atomic actions [9] in a usable manner. Ada 95 does not support atomic actions directly but does provide a rich supply of language primitives which potentially can be used to implement the same functionality. In particular:

packages. Provide encapsulation and information hiding.

exceptions. Provide a basic termination model of exception handling.

protected objects. Provide a monitor-like communication mechanism.

asynchronous transfer of controls (ATC). Provides a mechanism by which one task can asynchronously obtain the attention of another.

tagged types. Provide the framework from within which object-oriented programming can be performed.

controlled types. Provide the mechanism by which finalization code can be associated with objects.

Packages and exceptions were available in Ada 83 and, therefore, will not be discussed further. However, protected objects, ATC, tagged types and controlled types are new and will be briefly described. For a full discussion on these aspects of Ada 95, see Burns and Wellings [20].

4.1 Protected Objects

A protected object in Ada 95 is similar in concept to a conditional critical region [21], [22] and a monitor [23], [24]. Data which is to be accessed with mutual exclusion is encapsulated in a protected object. This data can only be accessed by subprograms and entries also declared in the protected object. Execution of a procedure or an entry requires mutual exclusive access over the object. As with tasks in Ada, protected objects may be declared as instances of a type, or a single one-off protected object (of an anonymous type). For example, the following protected object is a single instance of an anonymous protected type which allows mutually exclusive access to shared data.

```
protected Shared_Data is

  -- operations on the shared data
  procedure Write(D : Data);
  procedure Read(D: out Data);

private
  The_Data : Data := Some_Initial_Value;
  -- the encapsulated data
end Shared_Data;

protected body Shared_Data is

  procedure Write(D : Data) is
  begin
    The_Data := D;
  end Write;

  procedure Read(D : out Data) is
  begin
    D := The_Data;
  end Read;
end Shared_Data;
```

The difference between a procedure and an entry in a protected object is as follows. A procedure simply provides mutual exclusive access to the data. If there is no other task active in the protected object, a call on the procedure will gain immediate access to the data. An entry has an associated guard (called a barrier). A call to a guarded entry will only be allowed if the guard evaluates to true and there is no other task active in the protected object. If the guard is false, the task will be placed on a queue associated with the entry. The following shows the above reader/writer example when a write must initialize the data before it is read.

```
protected Shared_Data is

  procedure Write(D : Data);
  entry Read(D: out Data);

private
  The_Data : Data;
  Data_Available : Boolean := False;
end Shared_Data;

protected body Shared_Data is

  procedure Write(D : Data) is
  begin
    The_Data := D;
    Data_Available := True;
    -- indicate that the data is available
  end Write;

  entry Read(D : out Data) when Data_Available is
  begin
    D := The_Data;
  end Read;
end Shared_Data;
```

Inside a protected entry, the call can be *requeued* back onto the same entry or another entry of the same (or different) protected object.

4.2 ATC

The Ada 83 selective entry call facility is extended in Ada 95 to allow a task to execute a section of code whilst it is waiting for the entry (or timeout) to occur. If the code finishes before the entry call is accepted (or the timeout expires) then the call (or timeout) is cancelled. If the call is accepted (or timeout expires) before the section of code finishes then the execution of the code is aborted.

The following illustrates the syntax:

```
...
select
  Trigger.Event; -- trigger is a protected object
  -- optional sequence of statements to be
  -- executed after the event has been received
then abort
  -- abortable sequence of statements
end select;
```

4.3 Tagged Types and Object-Oriented Programming

Ada supports object-oriented programming through two complimentary mechanisms which provide type extensions and dynamic polymorphism: tagged types and class-wide types.

In Ada, a new type can be created from an old type and some of the properties of the type changed using *derived* types. For example, the following declares a new type called **Setting** which has the same properties as the **Integer** type but a restricted range. **Setting** and **Integer** are distinct and cannot be interchanged:

```
type Setting is new Integer range 1 .. 100;
```

New operations manipulating **Setting** can be defined; however no new components can be added. Tagged types remove this restriction and allow extra components to be added to a type. Any type that might potentially be extended in this way must be declared as a tagged type. Because extending the type inevitably leads to the type becoming a record, only record types (or private types which are implemented as records) can be tagged.

Tagged types provide the mechanism by which types can be extended incrementally. The result is that a programmer can create a hierarchy of related types. Other parts of the program may now wish to manipulate that hi-

erarchy for their own purposes without being too concerned which member of the hierarchy it is processing at any one time. Ada is a strongly typed language and, therefore, a mechanism is needed by which an object from any member of the hierarchy can be passed as a parameter.

Class-wide programming is the technique which enables programs to be written which manipulate families of types. Associated with each tagged type, `T`, there is a type `T'Class` which comprises all the types which are in the family of types starting at `T`. If an operation is called with a parameter whose type is class-wide, then run-time dispatching occurs to the appropriate operation for the associated actual type.

An object is typically defined by means of a package containing a tagged type and its primitive operations.

```
package Objects is
  type Obj_Type is tagged limited private;

  procedure Op1(O : in out Obj_Type);
  procedure Op2(O : in out Obj_Type);

  procedure Class_Wide_Op(O : in Obj_Type'Class);

private
  type Obj_Type is tagged limited
    record
      ...
    end record;
end Objects;
```

The object can be extended (possibly by a child library package—which allows access to the private part in the parent's declaration).

```
package Objects.Extended is
  type Extended_Type is new Obj_Type with private;

  procedure Op1(O : in out Extended_Type);
  procedure Op2(O : in out Extended_Type);

private
  type Extended_Type is new Obj_Type with
    record
      ...
    end record;
end Objects.Extended;
```

4.4 Controlled Types

Further support for object-oriented programming is provided by *controlled* types. With these types it is possible to define subprograms that are called (automatically) when objects of the type:

- are created—*initialize*;
- cease to exist—*finalize*;
- are assigned a new value—*adjust*.

To gain access to these features, the type must be derived from `Controlled`, a predefined type declared in the library package `Ada.Finalization`; that is, it must be part of the `Controlled` class hierarchy. The package `Ada.Finalization` defines procedures for `Initialize`, `Finalize`, and `Adjust`. When a type is derived from `Controlled` these procedure may be overridden. As objects typically cease to exist when they go out of scope, the exiting of a block may involve a number of calls of `Finalize`. A more restricted type `Limited_Controlled` provides initialization and finalization only.

5 Representing Recoverable Atomic Actions in Ada 95

In Section 2.1, the requirements for atomic actions were defined. These are now briefly reviewed to evaluate Ada 95 as a potential language for implementing software fault tolerance techniques.

1) *Well-defined boundaries*. Each atomic action can be encapsulated in one or more Ada packages and hence the side boundaries of each action are well-defined. Subprograms (procedures and functions) in the package interfaces can be used to provide the start and end points for each task.
2) *Indivisibility*. Protected objects provide the mechanisms with which the indivisibility property of an action can be implemented. Protected entries can be used to provide the required synchronization on exit from the action.
3) *Nesting*. Nested actions can be supported by implementing the actions as abstract data types.
4) *Concurrency*. Concurrency between the execution of atomic actions is provided by the concurrency between tasks. Groups of tasks which do not share an action will automatically execute concurrently.
5) *Recovery*. Backward and forward recovery can be programmed using a combination of protected objects, ATC, and exceptions.

The above points suggest that Ada 95 does have the appropriate mechanisms to facilitate the implementation of software fault tolerance techniques. The remainder of this section explores in detail how these mechanisms can be used. The overall approach is illustrated in Fig. 2.

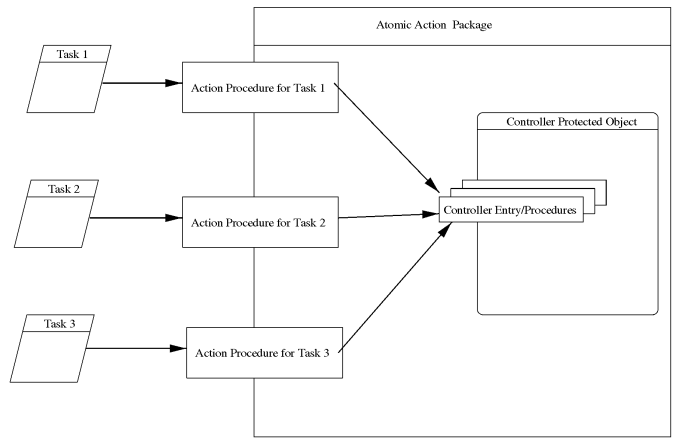


Fig. 2. An Ada 95 framework for implementing atomic actions.

5.1 Simple Actions

To show how atomic actions can be programmed in Ada, consider first a simple non-nested action with no recovery between, say, three tasks. The action is encapsulated in a package with three visible procedures, each of which is called by the appropriate task. It is assumed that no tasks are aborted and that there are no deserter tasks.

```
package Simple_Action is
  procedure T1(Params : Param); -- called by
                               -- Task 1
  procedure T2(Params : Param); -- called by
                               -- Task 2
  procedure T3(Params : Param); -- called by
                               -- Task 3
end Simple_Action;
```

The body of the package automatically provides the well-defined boundary, so all that is required is to provide the indivisibility. A protected object, **Action_Controller**, can be used for this purpose (it provides a similar function to the Coordinated Atomic Action manager introduced by Randell et al. [25], [26]). The package's visible procedures call the appropriate entries in the protected object. Four entries are sufficient. Each component of the action calls its own individual entry to indicate that it has arrived. On finishing its component, it calls the **Finished** entry.

```
package body Simple_Action is
  protected Action_Controller is
    entry First; -- called by T1
    entry Second; -- called by T2
    entry Third; -- called by T3
    entry Finished; -- called by all tasks
  private
    First_Here : Boolean := False;
    Second_Here : Boolean := False;
    Third_Here : Boolean := False;
    Release : Boolean := False;
  end Action_Controller;

  -- definition of local protected objects for
  -- controlling resources
protected body Action_Controller is separate;

procedure T1(Params) is
begin
  Action_Controller.First;
  -- acquire resources
  -- the action itself, communicates with tasks
  -- executing T2
  -- and T3 via resources
  Action_Controller.Finished;
  -- release resources
end T1;

-- similar for second and third task
begin
  -- any initialization of local resources
end Simple_Action;
```

The implementation of **Action_Controller** is given below.

```
separate(Simple_Action)
protected body Action_Controller is
    entry First when not First_Here is
    begin
      First_Here := True;
    end First;

    entry Second when not Second_Here is
    begin
      Second_Here := True;
    end Second;

    entry Third when not Third_Here is
    begin
      Third_Here := True;
    end Third;

    entry Finished when Release or Finished'Count
       = 3 is
    begin
      if Finished'Count = 0 then
        Release := False;
        First_Here := False;
        Second_Here := False;
        Third_Here := False;
      else
        Release := True;
      end if;
    end Finished;
  end Action_Controller;
```

The barriers of the entries **First**, **Second**, and **Third** ensure that only three tasks can be active in the action at any one time. Only when all three tasks have called the **Finished** entry is the barrier lowered and all tasks released. The Boolean **Release** is used to program the required release conditions on **Finished**. The first two calls on **Finished** will be blocked as both parts of the barrier expression are false. When the third call comes, the **Count** attribute will become three; the barrier comes down and one task will execute the entry body. The **Release** variable ensures that the other two tasks are both released. The last task to exit must ensure that the barrier is raised again.

Note that Ada's task identifiers can be used if it is necessary to validate the identity of each task performing the action components.

In the following sections, it will be assumed that only those tasks participating in the action use the package implementing the action and that each task will only call its associated operation (and no other).

5.2 Backward Error Recovery

In this section, the Ada ATC facility and exception handling are used to implement backward error recovery. Any scheme based on backward error recovery requires the use of some form of recovery cache. This section assumes the existence of the following generic package for saving and restoring a task's variables. It is assumed that the underlying Ada implementation and run-time are fault free, and therefore the strong typing provided by Ada will ensure that the Ada program itself remains viable.

```
generic
  type Data is private;
package Recovery_Cache is
  procedure Save(D : in Data);
  procedure Restore(D : out Data);
end Recovery_Cache;
```

Consider three Ada tasks which wish to enter into a recoverable atomic action. Each will call their appropriate procedure in the package given below.

```
package Conversation is

  procedure T1(Params : Param); -- called by
                               -- task 1
  procedure T2(Params : Param); -- called by
                               -- task 2
```

```
  procedure T3(Params : Param); -- called by
                               -- task 3

  Atomic_Action_Failure : exception;

end Conversation;
```

The body of the package encapsulates the action and ensures that only communication between the three tasks is allowed during the conversation.[1] The **Controller** protected object is responsible for propagating any error condition noticed in one task to all tasks, saving and restoring any persistent data in the recovery cache, and ensuring that all tasks leave the action at the same time. It contains three protected entries and a protected procedure.

- The **Wait_Abort** entry represents the asynchronous event on which the tasks will wait whilst performing their part of the action (the first call, indicates the start of a new action and ensures that the **Controller** saves any persistent data in the recovery cache—the requeue facility is used to place the first task back on the queue). The entry is also responsible for restoring any persistent data (defined within the package body).
- Each task calls **Done** if it has finished its component of the action without error. Only when all three tasks have called **Done** will they be allowed to leave.
- Similarly, each task calls **Cleanup** if it has had to perform any recovery.
- If any task recognizes an error condition (either because of a raised exception or the failure of the acceptance test), it will call **Signal_Abort**. This will set the flag **Killed** to true, indicating that the tasks must be recovered.

Note, that as backward error recovery will be performed, the tasks are not concerned with the actual cause of the error. When **Killed** becomes true, all tasks in the action receive the asynchronous event. Once the event has been handled, all task must wait on the **Cleanup** entry so that they all can terminate the conversation module together.

```
with Recovery_Cache;
package body Conversation is

  Primary_Failure, Secondary_Failure,
          Tertiary_Failure: exception;
  type Module is (Primary, Secondary, Tertiary);

  package Persistent_Cache is new Recovery_Cache
        (...);
  -- for any persistent data to be retained
  -- between conversations

protected Controller is
  entry Wait_Abort;
  entry Done;
  entry Cleanup;
  procedure Signal_Abort;
private
  Killed : Boolean := False;
  Releasing_Done : Boolean := False;
  Releasing_Cleanup : Boolean := False;
  Informed : Integer := 0;
  New_Conversation : Boolean := True;
end Controller;

-- any local protected objects for communication
-- between actions
protected body Controller is
  entry Wait_Abort when Killed or New_Conversation
        is
  begin
    if New_Conversation then
      -- save any persistent data in the recovery
      -- cache
      Persistent_Cache.Save(...);
      New_Conversation := False;
      requeue Wait_Abort with abort;
      --no return to here
    end if;

    -- only executed when Killed = True
    Informed := Informed + 1;
    if Informed = 3 then
      Killed := False;
      Informed := 0;
      Persistent_Cache.Restore(...);
      -- restore any persistent data
    end if;
  end Wait_Abort;

  procedure Signal_Abort is
  begin
    Killed := True;
  end Signal_Abort;

  entry Done when Done'Count = 3 or
        Releasing_Done is
  begin
    if Done'Count > 0 then
      Releasing_Done := True;
    else
      Releasing_Done := False;
      New_Conversation := True;
    end if;
  end Done;

  entry Cleanup when Cleanup'Count = 3 or
        Releasing_Cleanup is
  begin
    if Cleanup'Count > 0 then
      Releasing_Cleanup := True;
    else
      Releasing_Cleanup := False;
      New_Conversation := True;
    end if;
  end Cleanup;
end Controller;

  procedure T1(Params : Param) is separate;
  procedure T2(Params : Param) is separate;
  procedure T3(Params : Param) is separate;
end Conversation;
```

The code for each task is contained within a single procedure: e.g., **T1**. Within such a procedure, three attempts are made to perform the action. If all attempts fail, the exception **Atomic_Action_Failure** is raised. Each attempt is surrounded by a call that saves the state and restores the

1. In practice, this might be difficult to ensure because of Ada's scope rules. One way of increasing the security would be to require that the **Conversation** package is at the library level and its body only references pure (state free) packages.

state (if the attempt fails). Each attempt is encapsulated in a separate local procedure (**`T1_Primary`**, etc.), which contains a single 'select and then abort' statement to perform the required protocol with the controller. The recovery cache is used by each task to save its local data.

```
separate Conversation)
procedure T1(Params : Param) is
  procedure T1Primary is
  begin
    select
      Controller.Wait_Abort; -- triggering event
      Controller.Cleanup; -- wait for all to finish
      raise Primary_Failure;
    then abort
      begin
        -- code to implement atomic action,
        -- the acceptance test might raise
        -- an exception
        if Accept_Test = Failed then
          Controller.Signal_Abort;
        else
          Controller.Done; -- signal completion
        end if;
      exception
        when others =>
          Controller.Signal_Abort;
      end;
    end select;
  end T1_Primary;

  procedure T1_Secondary is ... ;
  procedure T1_Tertiary is ... ;

  package My_Cache is new Recovery_Cache(..);
        -- for local data
begin
  My_Cache.Save(..);
  for Try in Module loop
    begin
      case Try is
        when Primary => T1_Primary; exit;
        when Secondary => T1_Secondary; exit;
        when Tertiary => T1_Tertiary;
      end case;
    exception
      when Primary_Failure =>
        My_Cache.Restore(..);
      when Secondary_Failure =>
        My_Cache.Restore(..);
      when Tertiary_Failure =>
        My_Cache.Restore(..);
        raise Atomic_Action_Failure;
      when others =>
        My_Cache.Restore(..);
        raise Atomic_Action_Failure;
    end;
  end loop;
end T1;

-- similarly for T2 and T3
```

Fig. 3. illustrates a simple state transition diagram for a participating task in an Ada 95-based conversation.

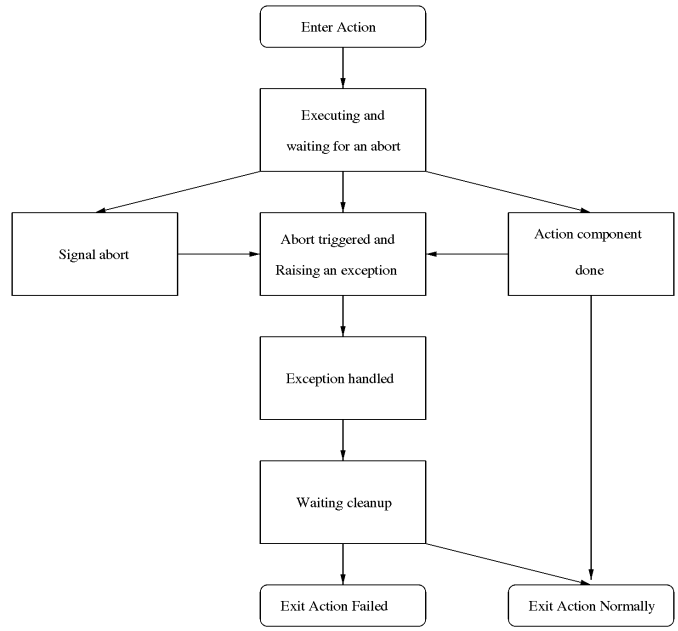


Fig. 3. Simple state transition diagram for a conversation.

5.3 Forward Error Recovery

Ada's ATC facility can be used with exceptions to implement atomic actions with forward error recovery between concurrently executing tasks. Consider again the following package for implementing an atomic action between three tasks.

```
package Action is

  procedure T1(Params : Param); -- called by
                               -- task 1
  procedure T2(Params : Param); -- called by
                               -- task 2
  procedure T3(Params : Param); -- called by
                               -- task 3

  Atomic_Action_Failure : exception;

end Action;
```

As with backward error recover, the body of the package encapsulates the action and ensures that only communications between the three tasks are allowed. The **`Controller`** protected object is responsible for propagating any exception raised in one task to all tasks, and for ensuring that all leave the action at the same time.

```
with Ada.Exceptions;
use Ada.Exceptions;
package body Action is

  type Vote_T is(Commit, Aborted);
  protected Controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    entry Cleanup (Vote: Vote_T; Result :
          out Vote_T);
    procedure Signal_Abort(E: Exception_Id);
  private
    entry Wait_Cleanup(Vote: Vote_T; Result :
          out Vote_T);
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
```

```
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_T := Commit;
    Informed : Integer := 0;
  end Controller;

-- any local protected objects for
-- communication between actions
protected body Controller is
  entry Wait_Abort(E: out Exception_Id) when
        Killed is
  begin
    E := Reason;
    Informed := Informed + 1;
    if Informed = 3 then
      Killed := False;
      Informed := 0;
    end if;
  end Wait_Abort;

  entry Done when done'Count = 3 or
        Releasing_Done is
  begin
    if Done'Count > 0 then
      Releasing_Done := True;
    else
      Releasing_Done := False;
    end if;
  end Done;

  entry Cleanup (Vote: Vote_T; Result:
        out Vote_T) when True is
  begin
    if Vote = Aborted then
      Final_Result := Aborted;
    end if;
    requeue Wait_Cleanup with abort;
  end Cleanup;

  procedure Signal_Abort(E: Exception_Id) is
  begin
    Killed := True;
    Reason := E;
  end Signal_Abort;

  entry Wait_Cleanup (Vote: Vote_T; Result:
        out Vote_T)
        when Wait_Cleanup'Count = 3 or
             Releasing_Cleanup is
  begin
    Result := Final_Result;
    if Wait_Cleanup'Count > 0 then
      Releasing_Cleanup := True;
    else
      Releasing_Cleanup := False;
      Final_Result := Commit;
    end if;
  end Wait_Cleanup;
end Controller;
procedure T1(Params: Param) is
  X : Exception_Id;
  Decision : Vote_T;
  begin
    select
      Controller.Wait_Abort(X);
            -- triggering event
      Raise_Exception(X);
             -- raise common exception
    then abort
      begin
        -- code to implement atomic action
        Controller.Done; -- signal completion
      exception
        when E: others =>
          Controller.Signal_Abort
                (Exception_Identity(E));
      end;
    end select;
  exception
    -- if  any exception is raised during the
    -- action
    -- all tasks must participate in the recovery
    when E: others =>
      -- Exception_Identity(E) has been raised
      -- in all tasks

      -- handle exception
      if Handled_Ok then
        Controller.Cleanup(Commit, Decision);
      else
        Controller.Cleanup(Aborted, Decision);
      end if;
      if Decision = Aborted then
        raise Atomic_Action_Failure;
      end if;
  end T1;

  procedure T2(Params : Param) is ...;
  procedure T3(Params : Param) is ...;

end Action;
```

Each component of the action (**T1**, **T2**, and **T3**) has identical structure. The component executes a select statement with an abortable part. The triggering event is signaled by the **Controller** protected object if any component indicates that an exception has been raised and not handled locally in one of the components. The abortable part contains the actual code of the component. If this code executes without incident, the **Controller** is informed that this component is ready to commit the action. If any exceptions are raised during the abortable part, the **Controller** is informed and the identity of the exception passed. Note that, unlike backward error recovery (given in the previous section), here the cause of the error must be communicated.

If the **Controller** has received notification of an unhandled exception, it releases all tasks waiting on the **Wait_Abort** triggering event (any task late in arriving will receive the event immediately it tries to enter into its select statement). The tasks have their abortable parts aborted (if started), and the exception is raised in each task by the statement after the entry call to the controller. If the exception is successfully handled by the component, the task indicates that it is prepared to commit the action. If not, then it indicates that the action must be aborted. If any task indicates that the action is to be aborted, then all tasks will raise the exception **Atomic_Action_Failure**. Fig. 4 shows the approach using a simply state transition diagram.

Fig. 4 illustrates that it is possible to program atomic actions with forward error recovery in Ada. However, only the first exception to be passed to the **Controller** will be raised in all tasks. It is not possible to get concurrent raising

of exceptions, as any further exception raised in an abortable part is lost when it is aborted.

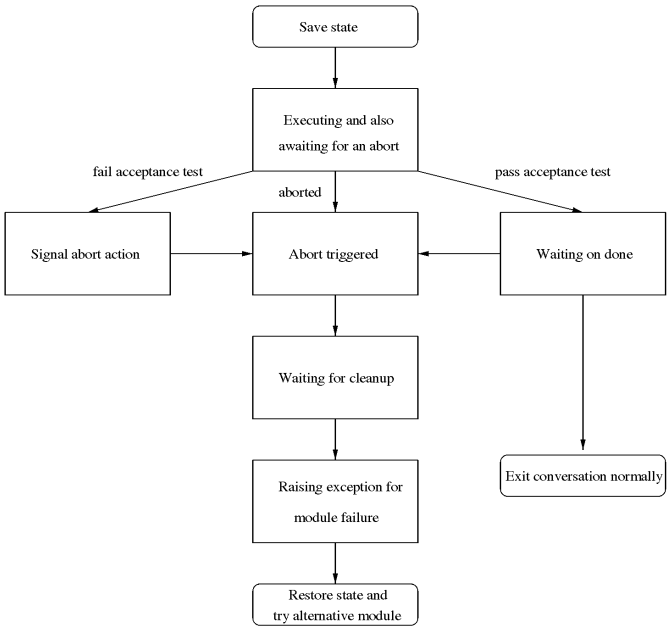


Fig. 4. Simple state transition diagram illustrating forward error recovery.

5.4 The Deserter Problem and Task Aborts

So far, it has been assumed that all expected tasks arrive and leave the action. If a task fails to arrive, all tasks will be blocked trying to leave. To solve the *deserter* problem, it is necessary to know how many tasks have entered the action. When all the tasks that have entered are ready to leave, the action can complete. A simple modification to the action controller protected object allows for this:

```
protected Controller is
  entry Wait_Abort(E: out Exception_Id);
  entry Done;
  entry Cleanup (Vote: Vote_T; Result :
        out Vote_T);
  procedure Signal_Abort(E: Exception_Id);
private
  entry Entered(E: out Exception_Id);
  entry Wait_Cleanup(Vote: Vote_T; Result :
        out Vote_T);
  Killed : Boolean := False;
  Releasing_Done : Boolean := False;
  Releasing_Cleanup : Boolean := False;
  Reason : Exception_Id;
  Final_Result : Vote_T := Commit;
  Active : Integer := 0;
end Controller;
```

A new private entry is provided called **Entered** and a count, **Active**, of the number of tasks active in the action. When a task calls the **Wait_Abort** entry as part of the ATC statement, the call is immediately accepted. The count incremented, and the call requeued on the **Entered** entry.

```
Entry Wait_Abort(E: out Exception_Id) when
      True is
begin
  Active := Active + 1;
  requeue Entered with abort;
end Wait_Abort;

entry Entered(E: out Exception_Id) when Killed is
begin
  E := Reason;
end Entered;
```

The guards on the **Done** and **Wait_Cleanup** entry are now simply changed to include **Active**, which is reset to zero when all tasks have finally finished the action:

```
entry Done when Done'Count = Active or
      Releasing_Done is
begin
  if Done'Count > 0 then
    Releasing_Done := True;
  else
    Releasing_Done := False;
    Active := 0;
  end if ;
end Done;

entry Wait_Cleanup (Vote: Vote_T;
      Result :out Vote_T)
      when Wait_Cleanup'Count = Active or
           Releasing_Cleanup is
begin
  Result := Final_Result;
  if Wait_Cleanup'Count > 0 then
    Releasing_Cleanup := True;
  else
    Releasing_Cleanup := False;
    Final_Result := Commit;
    Active := 0;
    Killed := False;
  end if ;
end Wait_Cleanup;
```

All that is now required is for the Action itself to contain timeouts on any synchronous communication which might block if a cooperating task is absent. If the communication was essential to the action, then an exception can be raised if the timeout expires.

One final problem to address is what happens if a task executing an atomic action is aborted by another task outside the action. To facilitate recovery in the action, it is necessary to use another Ada 95 facility called **Controlled** types. Objects of a controlled type can have (amongst other things) finalization routines defined. Hence, each action procedure has the following extra components:

```
type Abort_Recovery is new
        Finalization.Limited_Controlled with
             null record;
procedure Finalize;
```

Where, finalization is used to signal to the action controller that the action is to be finalized:

```
procedure Finalize is
begin
  Controller.Deregister;
end Finalize;
procedure T1 is
  ...
  Ar : Abort_Recovery;
begin
  ...
end T1;
```

If the caller of **T1** is aborted, the **Ar** controlled variable goes out of scope. However, before this can happen, the finalization procedure is called. Note that the **Finalize** routine is called *every* time the variable goes out of scope – irrespective of whether the action was aborted or not! Hence, the **Controller** itself must recognize when the action is being finalized prematurely:

```
protected body Controller is
  ...
  procedure Deregister is
  begin
    if Active /= 0 then -- premature finalization
      Killed := True;
      Releasing_Done := True;
      Releasing_Cleanup := True;
      Reason := Atomic_Action_Failure'Identity;
      Final_Result := Aborted;
    end if;
  end Deregister;
  ...
 end Controller;
```

If controlled types are used, it is possible to optimize the solution to the deserter problem by defining an **Initialize** procedure which calls the **Controller** to register its arrival. Furthermore, it is possible to use the **Deregister** procedure to establish the correct post conditions and hence simplify **Done** and **Wait_Cleanup**. This approach is illustrated in Section 5.6;

5.5 Nested Actions

Implementing nested actions in Ada 95 requires extensions to the above algorithms. The first is to convert the basic approach so that each atomic action is a type and, therefore, more than one instance can be created. This can easily be achieved by introducing the notion of an action identifier. For example, consider the implementation of actions with forward error recovery given in the previous section. The package specification now becomes:

```
package Action is

  type Action_Id is private;

  function New_Action return Action_Id;

  procedure T1(A : Action_Id, Other Params :
        Param);
  procedure T2(A : Action_Id, Other Params :
        Param);
  procedure T3(A : Action_Id, OtherParams :
        Param);

  Atomic_Action_Failure : exception;

private
  type Action T;
  type Action_Id is access Action T;
end Action;
```

As **Action_Id** is private, assignment and comparison are available. The implementation of the **Action_T** type is a record containing an instance of the **Controller** (which becomes a protected type) and instances of any persistent data and their controlling access protocols:

```
type Action_T is
  record
    C : Controller;
    ...
  end record;
```

The function **New_Action** creates a new instance of the **Action_T** and hence creates a new **Controller**.

The interface procedures use the **Action_Id** to generate the call to the correct controller. For example, if **A** is of type **Action_Id**:

```
A.C.Wait_Abort;
```

Once action types have been introduced then nested action can be called within the body of an action. This will produce the following Ada structure (where **Nested_Action** is a package implementing another atomic action and, therefore, contains its own **Action_Id**):

```
with Nested_Action;
package body Action
  N : Nested_Action.Action_Id :=
        Nested_Action.New_Action;

  procedure T1(A: Action_Id, OtherParams :
        Param) is
    X : Exception_Id;
    Decision : Vote_T;
  begin
    select
      A.C.Wait_Abort(X);
      RaiseException(X);
    then abort
      begin
      -- code to implement atomic action
      -- including
      Nested_Action.T1(N,...);
                -- nested action call
      A.C.Done; -- completion of outer action
    exception
      when E: others =>
        A.C.Signal_Abort(Exception_Identity(E));
    end;
   end select;
  exception
    when E: others =>
      -- Exception_Identity(E) has been raised
      -- in all tasks

      -- handle exception
      if Handled_Ok then
        A.C.Cleanup(Commit, Decision);
      else
        A.C.Cleanup(Aborted, Decision);
      end if;
      if Decision = Aborted then
        raise Atomic_Action_Failure;
      end if;
  end T1;
  ...
end Action;
```

If this code is expanded (to remove the explicit procedure **Nested_Action.T1**), then the following equivalent structure is obtained:

```
select
    A.C.Wait_Abort(...);
    Raise_Exception(...);
then abort
```

```
  begin
    -- code to implement atomic action
    -- including
    select
      N.C.Wait_Abort(..);
      Raise_Exception(..);
    then abort
      begin
        -- code to implement inner atomic action
        N.C.Done; -- signal completion
      exception
        when E: others =>
        N.C.Signal_Abort(Exception_Identity
             (E));
      end;
    end select;
    A.C.Done;    -- signal completion
  exception
    when E: others =>
      A.C.Signal_Abort(Exception_Identity
           (E));
  end;
end select;
```

When the outer action has an exception signalled, the outer 'then abort' sequence of code is aborted. This in turn will cause the inner action to be aborted and the finalization recovery (introduced in the previous section) will be invoked. Of course, a parameter needs to be added to the **Abort_Recovery** type to allow the correct controller to be called:

```
type Abort_Recovery(N : Action_Id) is new
         Finalization.Limited Controlled with
               null record;
procedure Finalize(Ar : in out Abort Recovery) is
begin
  Ar.N.C.Deregister;
end Finalize;
```

5.6 Object-Oriented Programming and Reusability

The action systems developed so far can easily be rewritten to make them extensible. Once written and debugged, they can be used in new systems.

For example, consider a basic package which provides only the action controller supporting forward error recovery. If the **Controller** protected type is placed in the private part of the package, the child packages can be written which will implement atomic actions for particular systems. The code is, therefore, reused.

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Finalization; use Ada.Finalization;
package Atomic_Action_Support is

  type Action_T is abstract tagged limited
       private;

  Atomic_Action_Failure : exception;

private
  type Vote_T is(Commit, Aborted);

  protected type Controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    entry Cleanup (Vote: Vote_T; Result :
          out Vote_T);
    procedure Signal_Abort(E: Exception_Id);
    entry Register;
    procedure Deregister;
  private
    entry Wait_Cleanup(Vote: Vote_T; Result :
        out Vote_T);
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_T := Commit;
    Active : Integer := 0;
    New_Action : Boolean := True;
    Normal_Termination : Boolean := False;
  end Controller;

  type Action_T is abstract tagged limited
    record
      C : Controller;
    end record;
end Atomic_Action_Support;
```

Notice in this example, that the **Action_T** is now a tagged type. This allows the type to be extended by the child packages.

The body of the package simply contains the body of the **Controller**. Actions can now be created by extending the **Action_T** type. An example of this is given in the next Section. The full package is given as an appendix.

6 An Example Action System

Consider an example of controlling the position of a three axis robot. The software consists of several tasks including a coordinate manager and three tasks controlling the robot itself (one for each axis). The coordinate manager task informs the other three tasks when a new position is required. The act of moving from one position to another is an atomic action; the rest of the system should only see the robot in one position or another.

The action support for the robot is a child package of the **Atomic_Action_Support** given in the previous section. The **Action_T** is extended to include a protected type used to communicate and synchronize between the four tasks. Only when the coordinate manager task has written a new position can the other tasks acquire it. The routines are added for each task.

```
package Atomic_Action_Support.Robot is

  type My_Action_T is new ActionT with private;

  type Coordinate is range 0 .. 180;
  type Coordinates is record
    X : Coordinate;
    Y : Coordinate;
    Z : Coordinate;
  end record;

  procedure Coord_Manager(A : access My_Action_T;
                          New_Pos : Coordinates);
  procedure X_Coord(A : access My_Action_T);
  procedure Y_Coord(A : access My_Action_T);
  procedure Z_Coord(A : access My_Action_T);

private
  protected type Shared_Coord is
```

```
    procedure Write_Coords(NewPos:
          Coordinates);
    entry Read_X(X : out Coordinate);
    entry Read_Y(Y : out Coordinate);
    entry Read_Z(Z : out Coordinate);
  private
    Next_Pos : Coordinates;
    New_Value : Boolean := False;
    X_Got : Boolean := False;
    Y_Got : Boolean := False;
    Z_Got : Boolean := False;
  end Shared_Coord;

  type My_Action_T is new Action_T with
    record
      Sc : Shared_Coord
    end record;

  type Abort_Recovery(N : access My_Action_T)
       is new
         Limited_Controlled with null record;

  procedure Finalize(Ar : in out Abort_Recovery);
  procedure Initialize(Ar :in out Abort_Recovery);

end Atomic_Action_Support.Robot;
```

Note, here Ada 95's access parameters are used. This avoids having to allocate the action's data dynamically, and allows run-time dispatching of operations to be used should the action be further extended.

The body of the package is given below. The structure of each interface procedure is similar to that given in previous sections.

```
package body Atomic_ActionSupport.Robot is

  procedure Finalize(Ar : in out Abort_Recovery) is
  begin
    Ar.N.C.Deregister;
  end Finalize;

  procedure Initialize(Ar : in out Abort_Recovery) is
  begin
    Ar.N.C.Register;
  end Initialize;

  protected body Shared_Coord is separate;

  procedure X_Coord(A : access My_Action_T) is
    X : Exception_Id;
    Decision : Vote_T;
    Next : Coordinate;
    Ar : Abort_Recovery(A);
  begin
    select
      A.C.Wait_Abort(X);
      RaiseException(X);
    then abort
      begin
        -- code to implement atomic action
        -- including
        A.Sc.Read_X(Next);
        -- move to new position

        A.C.Done; --signal completion
      exception
        when E: others =>
        A.C.Signal_Abort(Exception_Identity
             (E));
        end;
    end select;
  exception
      when E: others =>
        -- move back to the origin
        A.C.Cleanup(Aborted, Decision);
        raise Atomic_Action_Failure;
  end X_Coord;

  -- similarly for Y_Coord and Z_Coord

  procedure Coord_Manager(A : access
        My_Action_T;
        New_Pos : Coordinates) is
    X : Exception_Id;
    Decision : Vote_T;
    Ar : Abort_Recovery(A);
  begin
    select
      A.C.Wait_Abort(X);
      Raise_Exception(X);
    then abort
      begin
        -- code to implement atomic action
        -- including
        A.Sc.Write_Coords(New_Pos);
        A.C.Done; -- signal completion
      exception
        when E: others =>
          A.C.Signal_Abort(Exception_Identity
              (E));
      end;
    end select;
  exception
      when E: others =>
        A.C.Cleanup(Aborted, Decision);
        raise Atomic_Action_Failure
  end Coord_Manager;
end Atomic_Action_Support.Robot;
```

The body of the **Shared_Coord** protected object is

```
separate Atomic_Action_Support.Robot)
protected body Shared_Coord is

    procedure Write_Coords(New_Pos: Coordinates)
          is
    begin
      Next_Pos := New_Pos;
      New_Value := True;
    end Write_Coords;

    entry Read_X(X : out Coordinate) when
        New_Value is
    begin
      X := Next_Pos.X;
      X_Got := True;
      if Y_Got and Z_Got then
        X_Got := False;
        Y_Got := False;
        Z_Got := False;
        New_Value := False;
       end if;
    end Read_X;

    -- similarly for Read_Y and Read_Z
end Shared_Coord;
```

Finally the code for the four tasks can be given:

```
with Atomic_Action_Support.Robot;
use Atomic_Action_Support.Robot;
procedure Main is
  Robot_Action : aliased My_Action_T;
  -- the action, all four tasks must enter
  -- before any can leave

  type Dimension is(X, Y, Z);

  task Control;
  task type Axis(A : Dimension);

  X_D : Axis(X); -- the three Axis controller
                 -- tasks
  Y_D : Axis(Y);
  Z_D : Axis(Z);

  task body Control is
    Start : Coordinates := (0,0,0);
    Next : Coordinates;
begin
  Coord_Manager(Robot_Action'Access, Start);
  loop
    -- determine next position
    Coord_Manager(Robot_Action'Access, Next);
  end loop;
exception
  when Atomic_Action_Failure =>
    ...
 end Control;

task body Axis is
begin
  loop
    case(A) is
      when X => X_Coord(Robot_Action'Access);
      when Y => Y_Coord(Robot_Action'Access);
      when Z => Z_Coord(Robot_Action'Access);
    end case;
    -- perform any required operation at the
    -- new position
  end loop;
exception{
  when AtomicActionFailure =>
    ...
end Axis;

begin
   null;
end Main;
```

One criticism of this approach is that the actions undertaken by the clients involve a complex communication protocol with the controller. This protocol can, however, be abstracted out of the application code and put in the **Atomic_Action_Support** package [27].

7 Conclusion

Atomic actions are a powerful dynamic structuring technique that allow software fault-tolerant systems to be implemented. However, it is not clear how a programming language or operating system should support their application. No commercial programming language or operating system provides direct support. The Ada 95 programming language does, however, provide a rich set of mechanisms to aid the programming of concurrent and real-time systems. This paper has shown how these facilities can be used to implement all aspects of atomic actions.

The Ada facilities are impressive because each defines support for particular functionality:

- encapsulation,
- communication and synchronization,
- exceptions,
- asynchronous transfer of control,
- object-oriented programming
- finalization.

These are the fundamental building blocks which allow resuable atomic actions to be constructed. The ability to program atomic actions in Ada should lead to their increase use in the engineering of high integrity applications.

Appendix

In this appendix the full code for the final `Atomic_Action_Support` package is given.

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Finalization; use Ada.Finalization;
package Atomic_Action_Support is

  type Action_T is abstract tagged limited private;

  Atomic_Action_Failure : exception;

private
  type Vote_T is(Commit, Aborted);

  protected type Controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    entry Cleanup (Vote: Vote_T; Result : out
          Vote_T);
    procedure Signal_Abort(E: Exception_Id);
    entry Register;
    procedure Deregister;
  private
    entry Wait_Cleanup(Vote: Vote_T; Result : out
          Vote_T);
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_T := Commit;
    Active : Integer := 0;
    New_Action : Boolean := True;
    Normal_Termination : Boolean := False;
  end Controller;

  type Action_T is abstract tagged limited
    record
    C : Controller;
  end record;

end Atomic_Action_Support;
with Ada.Exceptions;
use Ada.Exceptions;
package body Atomic_Action_Support is

  -- any local protected objects for
  -- communication between actions

  protected body Controller is

    entry Wait_Abort(E: out Exception_Id)
```

```
        when Killed is
    begin
      E := Reason;
    end Wait_Abort;

    entry Done when done'Count = Active or
          Releasing_Done is
    begin
      Releasing_Done := True;
      New_Action := False;
      Normal_Termination := True;
    end Done;

    entry Cleanup (Vote: Vote_T; Result:
          out Vote_T) when True is
    begin
      if Vote = Aborted then
        Final_Result := Aborted;

      end if;

      requeue Wait_Cleanup with abort;
    end Cleanup;

    procedure Signal_Abort(E: Exception_Id) is
    begin
      Killed := True;
      Reason := E;
    end Signal_Abort;

    entry Wait_Cleanup (Vote: Vote_T; Result:
          out Vote_T)
          when Wait_Cleanup'Count = Active
          or Releasing_Cleanup is
    begin
      Result := Final_Result;
      Releasing_Cleanup := True;
      New_Action := False;
      Normal_Termination := True;
    end Wait_Cleanup;

    procedure Deregister is
    begin
      Active := Active - 1;
      if Active = 0 then- last one out
        Killed := False;
        Releasing_Done := False;
        Releasing_Cleanup := False;
        Final_Result := Commit;
        New_Action := True;
        Normal_Termination := False;
    elsif not Normal_Termination then
       -- premature finalization
      Killed := True;
      Releasing_Cleanup := True;
      Reason := Atomic_Action_Failure'
                Identity;
      Final_Result := Aborted;
    end if;
  end Deregister;
  entry Register when New_Action is
  begin
    Active := Active + 1;
  end Register;

end Controller;

end Atomic_Action_Support;
```

ACKNOWLEDGMENTS

The authors would like to thank Dr. Stuart Mitchell, Dr. John Robinson, and Dr. Sascha Romanovsky for their comments on an earlier draft of this paper. The research reported in this paper has been partially sponsored by the European-funded DeVa project.

REFERENCES

[1] T. Anderson and P.A. Lee, *Fault Tolerance Principles and Practice*, second edition. Prentice Hall Int'l, 1990.

[2] J.B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Comm. ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[3] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A Program Structure for Error Detection and Recovery," *Lecture Notes in Computer Science 16*, E. Gelenbe and C. Kaiser, eds., pp. 171–187. Springer-Verlag, 1974.

[4] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Software Eng.*, vol. 1, no. 8, pp. 811–826, 1986.

[5] D.B. Lomet, "Process Structuring, Synchronisation and Recovery Using Atomic Actions," *Proc. ACM Trans. Language Design for Reliable Software, SIGPLAN*, pp. 128–137, 1977.

[6] A. Burns and A.J. Wellings, *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1990.

[7] K.H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Trans. Software Eng.*, vol. 8, no. 3, pp. 189–197, 1982.

[8] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, vol. 8, no. 1, pp. 66–73, 1991.

[9] A. Burns and A.J. Wellings, "Programming Atomic Actions in Ada," *Ada Letters*, vol. 9, no. 6, pp. 67–79, 1989.

[10] A. Clematis and V. Gianuzzi, "Structuring Conversations in Operation/Procedure Oriented Programming Languages," *Computer Languages*, vol. 18, no. 3, pp. 153–168, 1993.

[11] A. Romanovsky and L. Strigini, "Backward Error Recovery via Conversations in Ada," *Software Eng. J.*, vol. 10, no. 6, pp. 219–232, 1995.

[12] "Ada 95 Reference Manual," ANSI/ISO/IEC-8652:1995, *Intermetrics*, 1995.

[13] B. Randell, P.A. Lee, and P.C. Treleaven, "Reliability Issues in Computing System Design," *ACM Computing Surveys*, vol. 10, no. 2, pp. 123–165, 1978.

[14] P. Jalote, "Atomic Actions in Concurrent Systems," UIUCDCS-R-85-1223, Dept. of Computer Science, Univ. of Illinois, 1985.

[15] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 220–232, 1975.

[16] S.T. Gregory and J.C. Knight, "A New Linguistic Approach to Backward Error Recovery," *Proc. 15th Ann. Int'l Symp. Fault-Tolerant Computing Digest of Papers*, pp. 404–409, 1985.

[17] A.M. Tyrrell and D.J. Holding, "Design of Reliable Software in Distributed Systems Using the Conversation Scheme," *IEEE Trans. Software Eng.*, vol. 12, no. 9, pp. 921–928, 1986.

[18] P. Jalote and R.H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," *14th Ann. Int'l Symp. Fault-Tolerant Computing Digest of Papers*, pp. 347–352, 1984.

[19] P. Jalote and R.H. Campbell, "Atomic Actions for Fault-Tolerance Using CSP," *IEEE Trans. Software Eng.*, vol. 12, no. 1, pp. 59–68, 1986.

[20] A. Burns and A J. Wellings, *Concurrency in Ada*. Cambridge Univ. Press, 1995.

[21] C.A.R. Hoare, "Towards a Theory of Parallel Programming," *Operating Systems Techniques*, pp. 61–71. Academic Press, 1972.

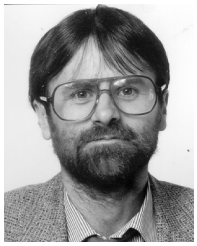
[22] P. Brinch-Hansen, "Structured Multiprogramming," *Comm. ACM*, vol. 15, no. 7, pp. 574–578, 1972.

[23] C.A.R. Hoare, "Monitors—An Operating System Structuring Concept," *Comm. ACM*, vol. 17, no. 10, pp. 549–557, 1974.

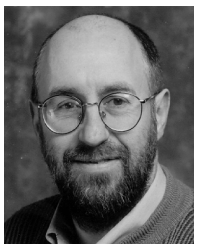
[24] P. Brinch-Hansen, *Operating System Principles*. Englewood Cliffs, N.J.: Prentice Hall, 1973.

[25] B. Randell et al., "From Recovery Blocks to Concurrent Atomic Actions," *Predictable Dependable Computing Systems*, B. Randell et al., eds. Springer-Verlag, 1995.

[26] Xu et al., "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery," *Digest of Papers, 25th Int'l Symp. Fault-Tolerant Computing*, pp. 499–508, 1995.
[27] A. Romanovsky, S. Mitchell, and A.J. Wellings, "On Programming Atomic Actions in Ada 95," *Proc. Ada Europe Conf.*, Springer-Verlag, 1997.



Andy Wellings is a professor of real-time systems in the Computer Science Department at the University of York, UK. He is interested in all aspects in the design and implementation of real-time dependable computer systems, on which he has authored/co-authored over 150 papers/reports. He is also the author of several books including *Concurrency in Ada* and *Real-Time Systems and Programming Languages*. He is European editor-in-chief for *the Computer Science Journal*, *Software-Practice and Experience*. Professor Wellings teaches courses in operating systems, real-time systems, and network and distributed systems.



Alan Burns has a personal chair in the Computer Science Department at the University of York, UK. His research activities have covered a number of aspects of real-time and safety critical systems including: requirements for such systems, the specification of safety and timings needs, systems architectures appropriate for the design process, the assessment of languages for use in the real-time safety critical domain, distributed operating systems, the formal specification of scheduling algorithms and implementation strategies, and the design of dependable user interfaces to safety critical applications. He has authored/co-authored over 200 papers/reports and eight books. Most of these are in the Ada or real-time area. His teaching activities include courses in operating systems, scheduling, and real-time systems.